Object- Oriented Modeling

Object-oriented modeling is a method that models the characteristics of real or abstract objects from application domain using classes and objects.

- **Objects**

  Software objects are modeled after real-world objects in that they too have **state** and **behavior**.

  - A software object maintains its state in one or more **variables** (**attributes**).
  - A software object implements its behavior with **methods** that manipulate these variables.

- **Messages**

  Software objects interact and communicate with each other by sending messages. When object A wants object B to perform one of B's methods, object A sends a message to object B.

Messages

Sometimes, the receiving object needs more information to know exactly what to do. This information is passed along with the message as **parameters**.

Message sending requires the following information:

- the object to which the message is addressed,
- the name of the method to perform,
- any parameters needed by the method.

The sending of a message can have any of the following effects:

- The state of the receiving object is changed.
- Some other actions are performed (including sending another messages to some objects).
- Some information is returned to the sending object.

Examples of Objects

Objects in a program correspond to objects from the application domain.

Information system of a bank: accounts, transactions, clients, other banks

Chess playing program: chess pieces, a chessboard, positions, moves, games, strategies

Action game: monsters, weapons, walls, doors, flying bullets, a score counter

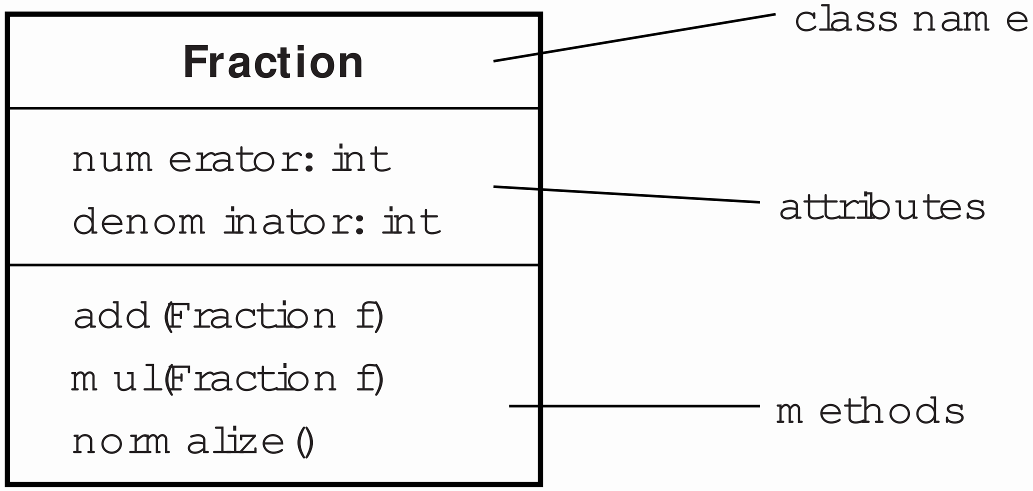
Drawing application: lines, rectangles, circles, arrows, text fields, line styles, line colors

GUI toolkit: windows, buttons, menus, menu items, icons

Class

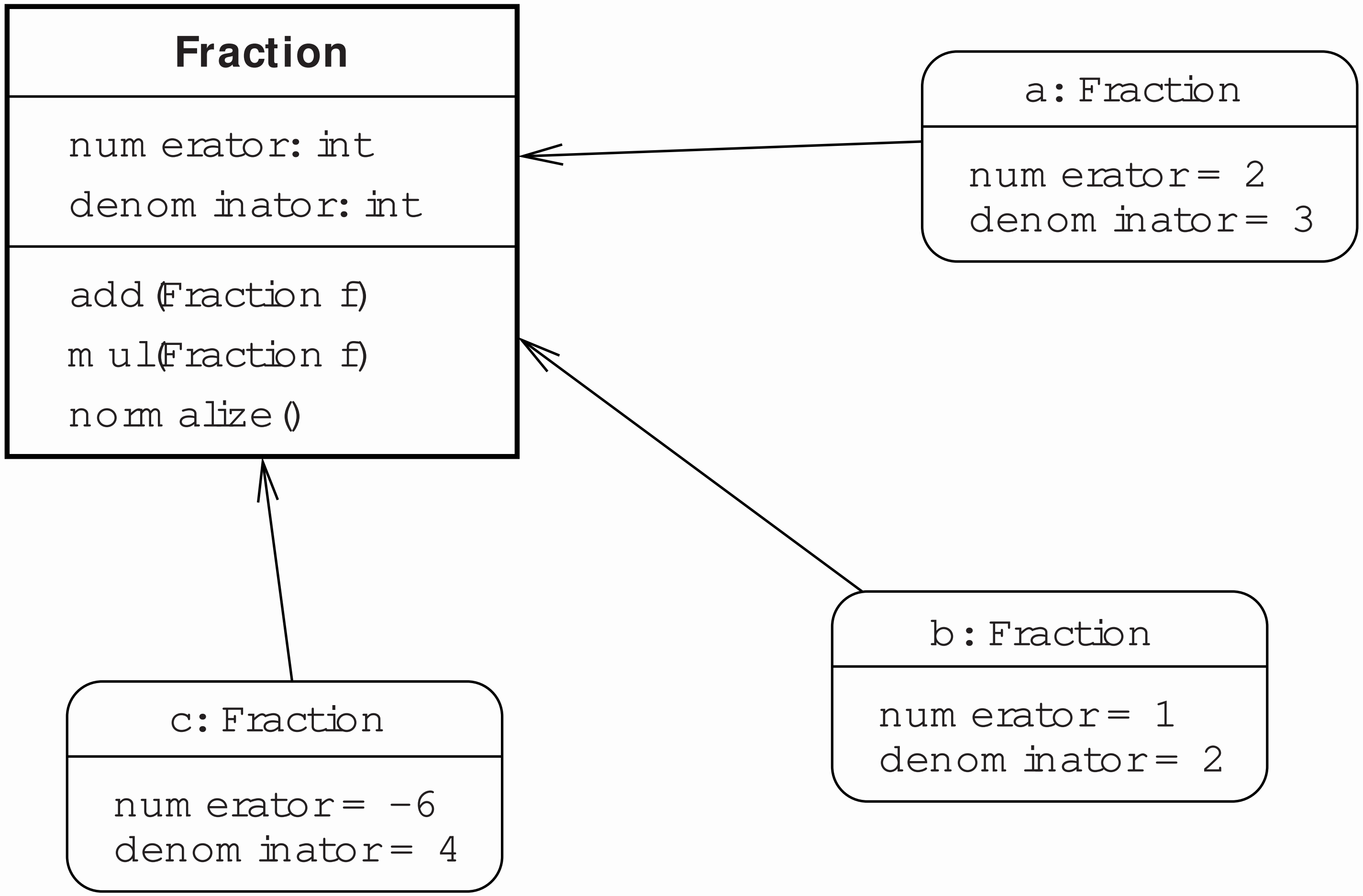
In the real world, many objects of the same kind exist. Using object-oriented terminology, the objects are **instances** of a **class**.
A class is prototype that defines variables and methods common to all objects of a certain kind.

Graphical representation of a class:



Instances

Objects are **instances** of the given **class**. Each object has its own **identity**.



Java Class

Java defines classes using the **class** keyword. Definition of a class may contain declarations of variables, definitions of methods or even nested classes.

The order of class members is not important.

```
class Fraction
{
    int  numerator;
    int  denominator;

    void mul(Fraction f)
    {
        numerator   *= f.numerator;
        denominator *= f.denominator;
    }
}
```

Note: Standard convention is that class names start with an upper-case letter and class member names (attributes and methods) start with a lower-case letter.

Java Objects

An instance of a class can be created using **new** operator. It allocates required space on the heap, provides initialization and returns reference to the newly created instance.

```
Fraction a = new Fraction();
```

The attributes of a object can be accessed using the reference to the object and the dot (.).

```
a.numerator = 2;
a.denominator = 3;
```

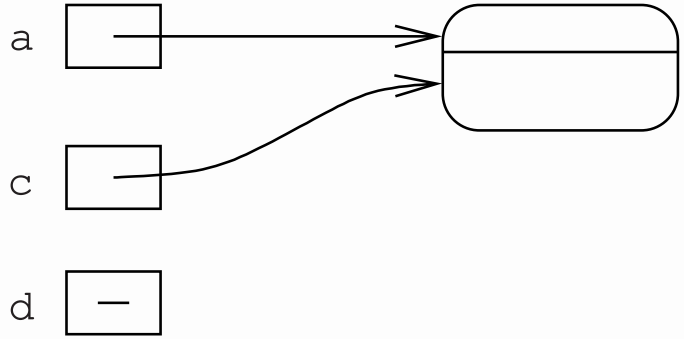
The methods of an object are accessed similarly.

```
Fraction b = new Fraction();
b.numerator = -6;
b.denominator = 4;

a.mul(b);
```

References

The references are just pointers to objects in memory. So after the following assignment the both a and c point to the same object.

```
Fraction c = a;
```



There is a special reference value **null** that denotes reference that does not point to any object.

```
Fraction d = null;
```

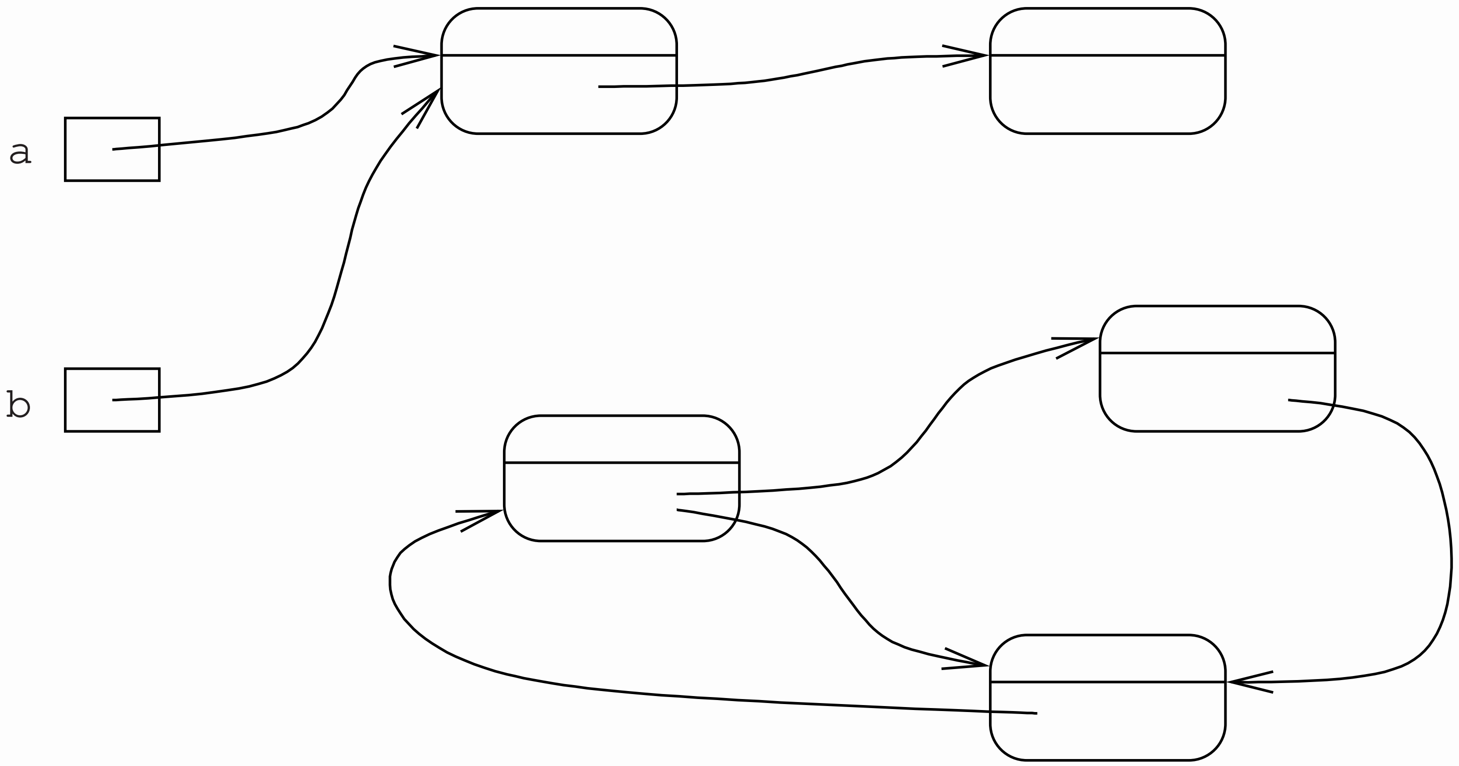
It is an error to access attributes or methods using a reference with the **null** value.

How To Destroy Objects?

There is no need to destroy objects explicitly in Java, since it uses automatic memory management – **garbage collector**.

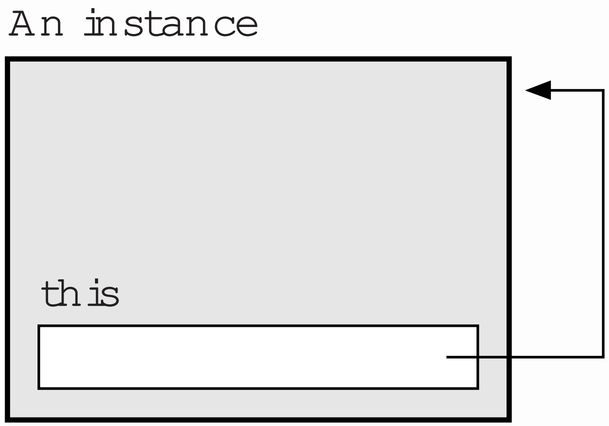
Whenever there is no reference to an object, the object can be destroyed and the memory used by this object is freed.

The garbage collector takes into account circular dependencies.



The `this` Keyword

The **this** keyword produces the reference to the object the method has been called for.



Examples (it is possible to omit **this** in most cases):

```
Fraction a, b;
   ...
this.numerator = 3;                 numerator = 3;
this.denominator = 2;               denominator = 2;
this.add(a);                        add(a);
b.mul(this);
```

Overloading of Methods

The methods in Java can be **overloaded**. This means that there can be more methods with the same name in a class. The methods must differ in the number and types of their parameters.

The method that is actually called is chosen depending on the number and types of parameters.

The return types of overloaded methods need not be all the same.

```
class Fraction {
      ...

    void add(int x) { ... }

    void add(int num, int den) { ... }

    void add(Fraction f) { ... }

      ...
}
```

Encapsulation

Some attributes and methods can be marked as **private**. Such attributes and methods can be accessed only from methods in the class where they are defined. An attempt to access them from methods in other classes produces a compile-time error.

Attributes and methods can be also marked as **public**. Such attributes and methods can be accessed from any other class.

```
class Fraction
{
    private int numerator;
    private int denominator;

    public void set(int num, int den) { ... }

    public int getNumerator() { ... }

    public int getDenominator() { ... }

      ...
}
```

Encapsulation (cont.)

Encapsulation (also **information hiding**) is the separation of the external aspects of an object (accessible from other objects) from the internal implementation details (which are hidden from other objects).

The implementation of an object can be changed without affecting the other parts of an application that use it.

In a purely object-oriented design the attributes of an object are always private and the only way to access them is through the methods that manipulate them.

The use of keywords **private** and **public** allows to promote encapsulation.

Note: If none of keywords **private** and **public** is used then the member can be accessed from other classes. However, there are differences between using and not using the keyword **public**. They will be discussed later. There is also a keyword **protected** that will be discussed later too.

Initialization of an Object

After the creation of a new object (using **new**), all its attributes are set to zero:

- numeric values (**int, long, char, float, …**) are set to 0
- boolean values are set to **false**
- references (to objects and to arrays) are set to **null**

It is possible to set attributes to some specified values using explicit initialization:

```
class Fraction
{
    int numerator = 0;
    int denominator = 1;

        ...
}
```

Constructors

A more elaborate initialization of an object can be implemented using **constructors**.

Constructor resemble methods, but there are some differences:

- The name of a class must be used as a name of a constructor.
- Constructors can not return values.
- A constructor can be invoked only in the time of the creation of an object (using **new**).

```
class Fraction {
        ...
    Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

```
Fraction a = new Fraction(3, 5);
a.add(new Fraction(1, 3));
```

Constructors (cont.)

Some additional remarks concerning constructors follow:

- Constructors may be overloaded, similarly as methods.
- Constructors can be marked as **public**, **protected** and **private**, similarly as methods.
- When no constructor is defined then the default constructor that does nothing is defined automatically, as if the following empty constructor would be put in the code:

```
class Fraction
{
    Fraction() { }
      ...
}
```

- If there is explicitly defined at least one constructor, the default empty constructor is not defined automatically.

Constructors (cont.)

Constructors can call other constructors. The keyword **this** can be used for this purpose. An invocation of another constructor can be used only as the first statement of a calling constructor's body.

```
class Fraction {

    Fraction() {
        this(0);
    }

    Fraction(int x) {
        this(x, 1);
    }

    Fraction(int num, int den) {
        numerator = num;
        denominator = den;
    }

     ...
}
```

Static Members

Variables and methods defined by a class can be of two types: **instance** and **class** (**static**). Class members are distinguished from instance ones by the **static** keyword.

- **Instance Members**
  - **Instance Variable**
    Any item of data that is associated with a particular object. Each instance of a class has its **own copy** of the instance variables defined in the class.
  - **Instance Method**
    Any method that is invoked with respect to an instance of a class.
- **Class Members**
  - **Class Variable**
    A data item associated with a particular class as a whole, not with particular instances of the class.
  - **Class Method**
    A method that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class.

Static Members (cont.)

- **Instance Variables and Methods**

  They can be accessed only using a reference to some object.

  ```
  obj.variable,  obj.method()
  ```

- **Class (Static) Variables and Methods**

  The keyword **static** is used to denote them. There is always exactly one copy of a static variable shared by all instances of the given class.

  ```
  static int count;
  static int getCount() {...}
  static void main(String[] args) {...}
  ```

  Class member are usually accessed using the class name.

  ```
  Test.count = 0;
  int c = Test.getCount();
  ```

  When they are accessed inside a given class, the class name can be omitted.

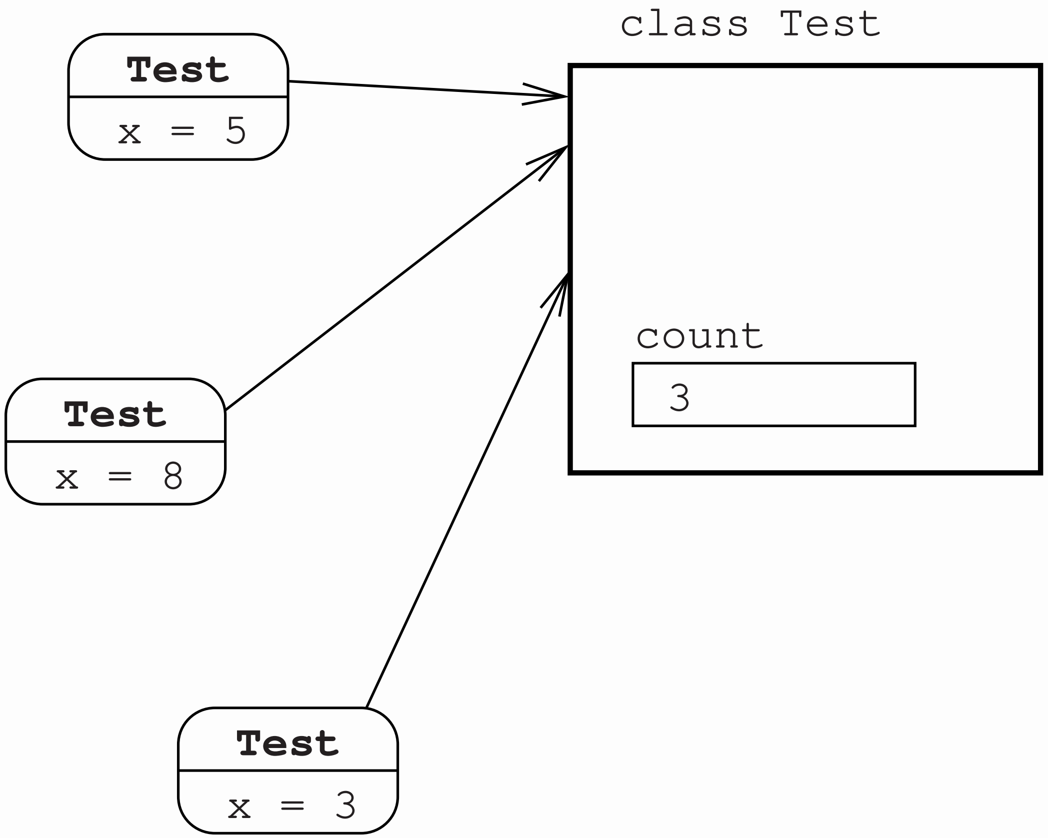
Static Members (cont.)

An example of use of static members:

```
class Test {

    private static int count = 0;
    private int x;

    public static int getCount() {
        return count;
    }

    public Test(int x) {
        this.x = x;
        count++;
    }

    public int getValue() {
        return x;
    }
}
```

Static Members (cont.)

There is exactly one copy of the static variable count in the memory. Each instance of the class Test has its own copy of the instance variable x.



Static Initializers

The static members can be initialized using **static initializer** of the form

```
static { ... }
```

The code may contain more than one static initializer. They are evaluated together with initializations of static variables in a textual order as they appear in a source file.

Static initializers are executed only once when the class is loaded into memory.

Static Initializers (cont.)

```
class StaticInitializerExample {

    static int x;

    static {
        x = 3;
        System.out.println(x);
    }

    static int y = 4;

    static {
        y = 1;
        System.out.println(y);
    }
     ...
}
```

produces the following output:

```
3
1
```